System Design Document for Wack

Viktor Franzen, Tobias Lindroth, Spondon Siddiqui, Alexander Solberg

November 21, 2018

Contents

1 Introduction

This document is supposed to give the reader an deeper understanding about the design of our project, "Wack".

1.1 Design goals

The main design goal of this project is to have an easily extended application that is loosely coupled. It should be easily tested and the application should also have a clear MVC-structure.

1.2 Definitions, acronyms, and abbreviations

GUI: Graphical User Interface; How the application looks.

MVC: Model-View-Controller; A design pattern that separates the logic from the view which makes it easy to reuse the model or change view.

UML: Unified Modeling Language; A general-purpose, modeling language used in the development of object oriented projects. It provides a standard way for the design of a system to be visualized.

JSON: JavaScript Object Notation; A language independent data format.

JavaFX: A Java standard GUI library.

FXML: An XML-based user interface markup language.

Channel: A chat group; A group where users of the application can join and send messages. Other users in the same group can see those messages.

2 System Architecture

This chat application will simulate an actual chat application with multiple computers. Hence, everything will be on the same computer, but using multiple clients.

When starting the program, there will be a login/sign up screen, from which the user can create clients. A client will open in a separate window and then the login/sign up screen can be used to create more clients.

In the top level of this application, we have the model, controllers, views and services packages and the main class (lies in "default package" in figure). There are no circular dependencies, and as seen in the figure below, the controller and services package both uses the model, but the model is not aware of who is using it.

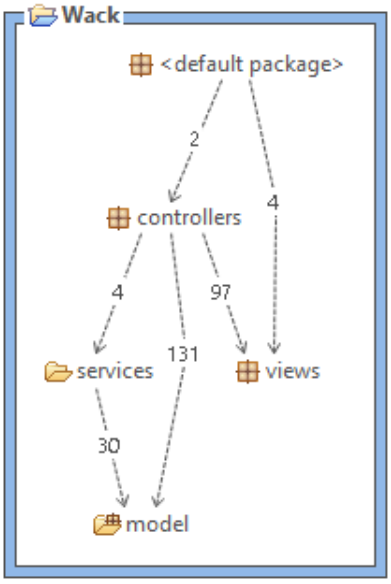


Figure 1: The top level

A MVC pattern has been used to structure the application. Hence, the application is split into model, view, and controller.

The model is not aware of what entity is using its data and logic. The separate views are completely unaware of the controllers handling them. The controllers act as a connection between the model and the view. It handles the inputs from the views by interacting with a facade towards the model and uses the information it receives to update the views. Further details about the connection between the model and the controllers as well as the controllers and the views, can be found in 2.3 and 2.4.

2.1 Subsystem Decomposition

As said before, the application is divided into 4 major packages which can be seen in figure 1. Model, Controllers, Views and Services

Model contains all the data and logic which is used to run the application.

Controller handles inputs and requests from view and communicates with the model.

Views handles the inputs from the user and forwards this to the controller.

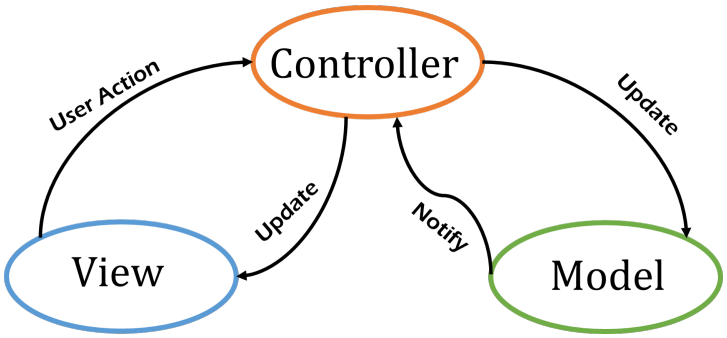


Figure 2: How mvc is used

Services are used to store data between sessions and to increase the security of the application.

The design model (Seen in the appendix) shows a UML diagram of how our different classes are connected. Here it can be seen that almost all classes depend on abstractions in the form of interfaces. It also shows that there is quite a loose coupling between different packages. An example is the yellow highlighted message package which attempts to be as modular as possible by having minimal coupling to other classes outside its package.

2.2 Model

The model component is responsible for all the data and logic of the program. The top level of the model contains the ChatFacade and four packages that are called server, client, chatcomponents and identifiers.

The model package at the top in figure 2.2 is really just the facade of the model, called ChatFacade. This is responsible for providing a facade for the model package, to minimize the dependencies on model implementations.

The client package is responsible for being able to receive updates and provide an interface for anyone that want to add themselves as listener to also receive the updates the client receives. The client will then forward it to all it's listeners.

The "chatcomponents" package is responsible for providing the objects that creates the structure of the chat system. This package is divided into three subpackages, channel, messages and user.

The channel package is responsible of providing a component where several users can send messages to each other.

The message package is responsible for a component that can be sent to others. The message package provides a factory for creating messages so other packages don't need to have dependencies on several different content classes.

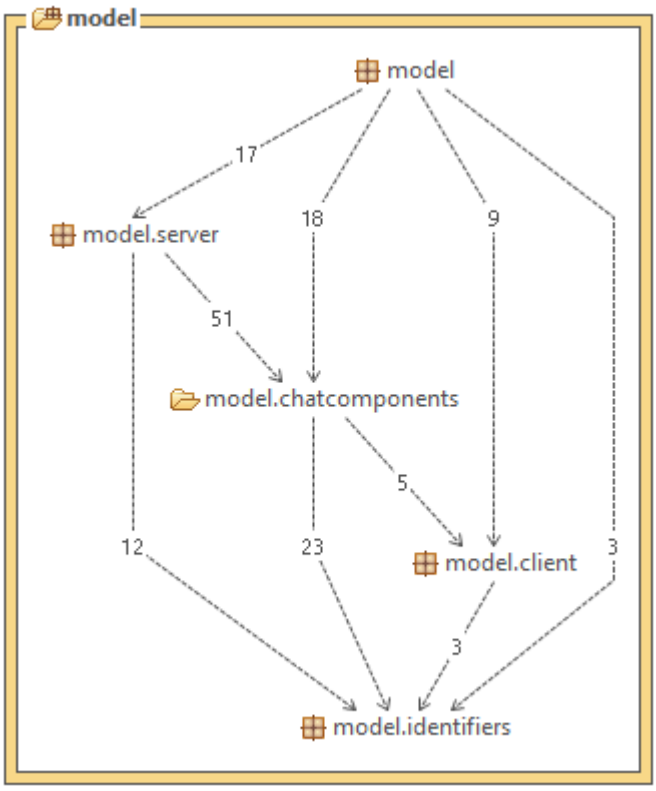


Figure 3: The top level

The user package contains all data about the user. This package is dependent on the client package because it needs to be able to tell clients when a channel has been updated. Hence, when the user receives an update it will forward these to all it's clients. A user can have multiply clients, to support the ability to log in on the same user on different devices(or in our case, different windows).

The server package is responsible for keeping track of all channels and users. Every time a channel or user is created it is added to the servers lists. The server is also responsible for saving the data when the program closes. To do this, it uses some given data handler that implements IDataHandler. It gives this datahandler certain data objects that stores all the channel or user data as strings. When initializing the server, the given datahandler is used to get these data objects back and then the server turns them into real channel and user objects. Read more about this in section 2.5

By breaking out the Identifiable and Recognizable interfaces into a separate

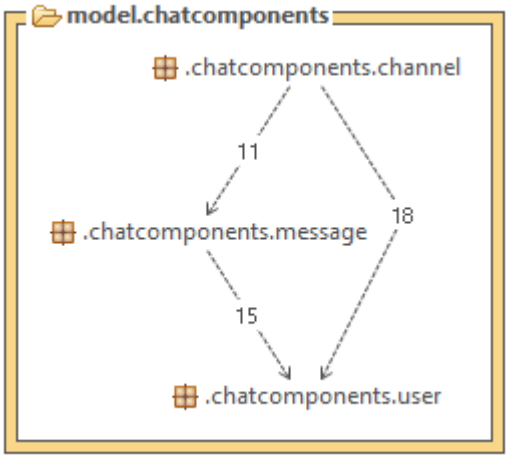
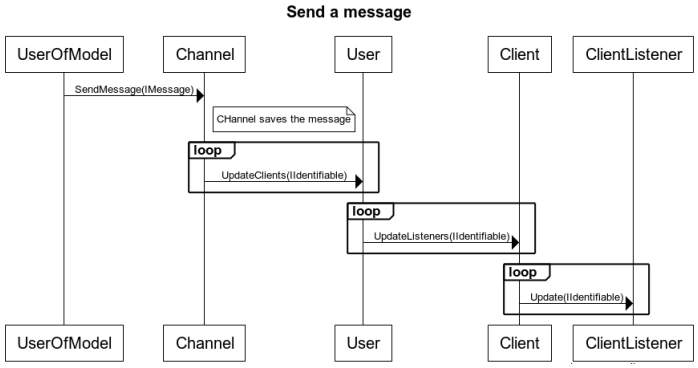


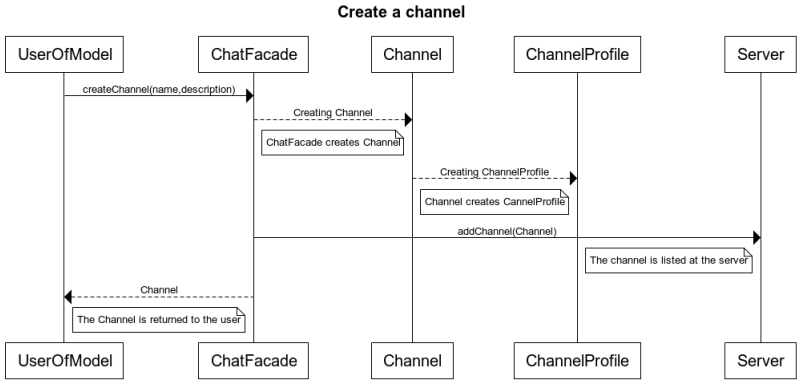
Figure 4: The chat components package

package called identifiers, responsible for providing abstractions of the chat components, the amount of dependencies has been reduced. Furthermore, there were some circular dependencies that have been removed completely due to the package decomposition

Below is two sequence diagrams that explains how some processes in the model works.



A channel is used to send a message. The channel saves the message, and notifies any members that the channel has been updated. The members then notifies any clients that a certain channel has been updated. The clients then notifies any client listeners about the update.



As seen in the figure, the ChatFacade will create the channel, add it to the Server, and then return it to be used.

2.3 Controllers

This component takes care of all the communication between the view and the model. It handles all input the view package receives, communicates with the model, and updates the view if needed. This package also listens to the model to be able to update when a channel of the represented client receives a message. To do this the observer pattern is used, and when the model tells this package that a new message has been sent, it will ask the model for the new message, and update the view according to the change.

The controllers are structured in the way that the login view has its own separate controller and the actual chat window has a main controller, that uses several other smaller controllers which each is responsible for a special part of the window.

The controllers are dependent on the model's ChatFacade and the following interfaces:

- IChannel
- IUser
- IClient
- IClientListener
- IIdentifiable
- IRecognizable

Since the model isn't depending on the controllers, it's really easy to change a controller. And since the dependency the controllers has on the model is quite

loose (depending only on interfaces and the facade), changes in the implementation of the model can be done without affecting the controllers.

Furthermore, the controllers connection with the view is also quite loosely coupled. The view package provides interfaces for every view component, as well as a corresponding controller interface, that controllers needs to implement to be able to be the controller of a view component.

A controller can then use the view components factory to create such a view and add themselves as the controller, hidden behind an interface. This means that the controller package only is depending on a factory and interfaces, which means that it's easy to change a controller.

If you were to switch one of the controllers, you would only need to make sure it would implement the controller interface of the view component.

2.4 Views

This component is responsible for providing the interface of the program. This view package uses JavaFx and provides several different view components and a corresponding controller interface to each component, and are to be used by any controller that implements the right interface. The views are built in fxml and are dynamically connected with the view classes.

As said in the controller package, the connection between the controllers and views is loose. Hence, if needed, it is quite easy to change the graphic library that are used. If you were to switch library you would only have to create new view components that implemented the view interfaces and used the controller interfaces. In figure 5 you can see the principal of how each controller and view are connected.

2.5 Services

The services package consists of two subpackages, password and datahandler.

The datahandler package is responsible for saving the programs data. To do this, it provides a concrete data handler class which implements an interface in the model. See figure 6. The controller package can then tell the model that this is the data handler to use. The model will then use this data handler, hidden behind the interface, to save data to a file and then later restore the data when its needed. When saving, the data handler is given certain data objects, that contains all the data of the components that are to be saved, as strings. When restoring, the model will ask the data handler to represent the data as these data objects and return them so that the model is able to recreate these data objects.

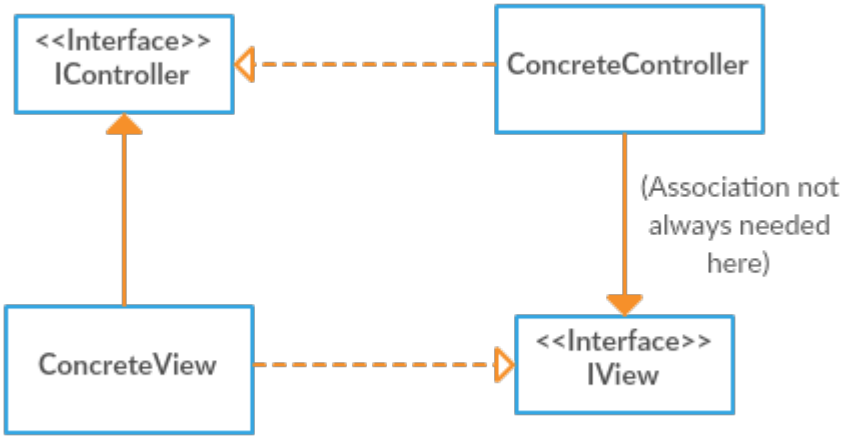


Figure 5: Dependencies between views and controllers

The passwordEncryption package is responsible for making sure to make the passwords encrypted, so when the data handler saves the password, it isn't in plain text. The connection with the model uses the same principal as the datahandler. It provides a concrete password encryption class that implements an interface in the model package. The controller package can then tell the model that this is the password encryptor to use. The model will then use this password encryptor, hidden behind the interface, to hash the users password.

3 Persistent Data Management

The application stores data by using JSON. When closing the application the program creates a JSON object where all the wanted data is stored. The object is then flushed using a FileWriter to a JSON-file. As the data is stored in a JSON file, it means that when we start the application again we can loop through the file and collect the saved data from the previous session.

In order to recreate the program's previous state the data is stored in two different Json files. These are explained below:

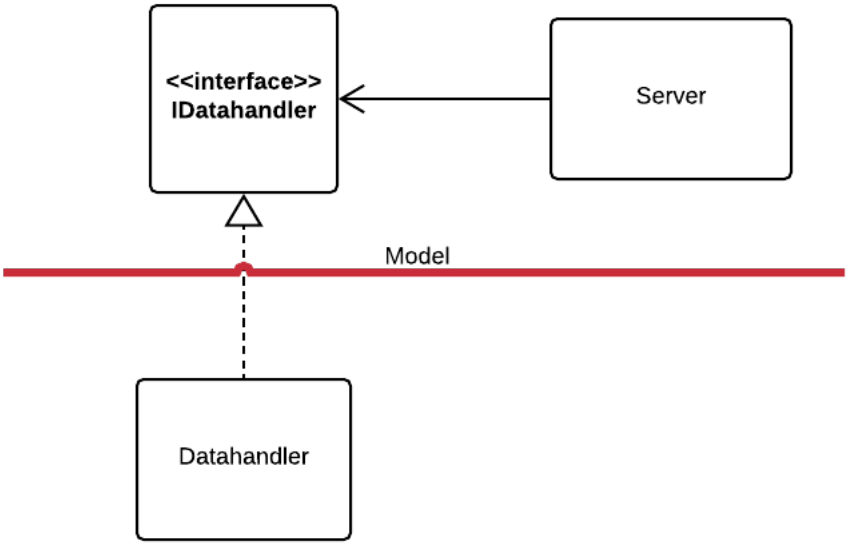


Figure 6: The dependency between the server and the datahandler

Channels are stored in one file, in the form of JSON-objects. These objects contain all the information that a channel controls. Each object contains a key and an associated value in the form of a String, JSONObject, or JSONArray. For example the users that are members of the channel are stored as JSONObjects within the channel JSONObject and the messages written in the channel are stored in a JSONArray.

Users are stored in another file, with each user stored in the form of JSON-objects. These objects contain all the information about the users and can be translated into objects that the model can use.

Images are stored locally in folders and are accessed by using the path to the image.

4 Access Control and Security

In the application there is different kind of roles which changes some of the things a user can do in the application.

User: The most basic role anyone using the application can have. As soon as you login you become a user. The only thing a user can do is join channels and see other channels.

Channel member: As soon as you join a channel you become a channel member. As a channel member you can write and read messages that are written inside the channel. You also can add other members to the channel.

Channel administrator: If you create a channel you automatically become a channel administrator. A channel administrator can change the channel image, kick members from the channel and also all the functionality a ordinary channel member.

5 Appendix

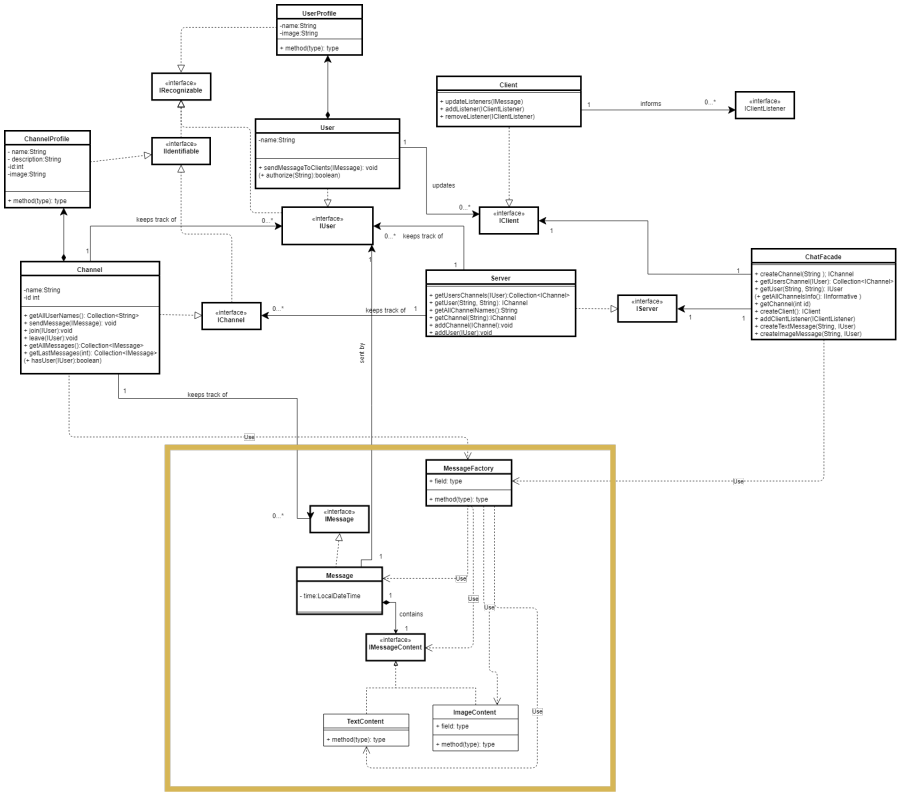


Figure 7: The design model